Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages

Umut Acar – Arthur Charguéraud – Mike Rainey

Max Planck Institute for Software Systems
Speedups with multicores

**Goal:** get good speedups from using several cores

**Obstacles:** lack of parallelism, memory wall, scheduling overheads
Granularity control

**Scheduling overheads:** they mainly depend on the number of tasks

Too many tasks:
→ large overheads

Not enough tasks:
→ limited parallelism

**Granularity control:** problem of finding the right size for parallel tasks

→ we propose a new approach to granularity control
  based on asymptotic complexity annotations
Importance of granularity control

Sequential code:

```c
int fibseq(int n) {
    if (n < 2) return 1;
    int a = fibseq(n-1);
    int b = fibseq(n-2);
    return a+b;
}
```

Parallel code:

```c
int fibpar(int n) {
    if (n < 2) return 1;
    spawn int a = fibpar(n-1);
    int b = fibpar(n-2);
    sync;
    return a+b;
}
```

compute fibonacci(45)

10 seconds on a single core

20 seconds on 42 cores

→ 1.8 billion parallel tasks created
→ per-task overhead of a few dozens memory accesses
Introduction of a cutoff value

Parallel code with cutoff value:

```c
int fibcut(int n) {
    if (n < cutoff)
        return fibseq(n)
    spawn int a = fibcut (n-1);
    int b = fibcut(n-2);
    sync;
    return a+b;
}
```

→ What is the right value to use as cutoff?
Execution time vs cutoff

Running fibpar(45) on 42 cores, using a work-stealing scheduler

many small tasks

few big tasks

good cutoffs

overheads start being amortized
parallelism now starts lacking
Selection of the cutoff value

- hard-coding a cutoff → non portable code
- trying all cutoffs (auto-tuning) → requires a tuning process

Our technique can automatically select a good cutoff at runtime
Amortizing task creation overheads

Idea: Assume that every fork costs $\tau$. If the cutoff value leads to tasks of size $\kappa \approx 100 \cdot \tau$, then the overheads are approximately equal to 1%.

Policy: tasks predicted to take less than $\kappa$ time are not parallelized

Remark: $\kappa$ depends on $\tau$, which depends on the hardware and the scheduler, but not on the algorithm, contrary to auto-tuning
Theory

**Brent's theorem:** (task creation overheads completely ignored)

\[ T_P \leq \frac{T_1}{P} + T_\infty \]

neglectable when lot of parallel available

**Our theorem:** (fork operation overhead = \( \tau \), sequentialize if exec time < \( \kappa \))

\[ T_P \leq \left( 1 + \frac{\tau}{\kappa} \right) \cdot \frac{T_1}{P} + \kappa \cdot T_\infty \]

we chose \( \kappa \) such that \( \tau/\kappa \approx 1\% \)

increased a lot but still remains neglectable
How to predict execution times?

In addition to:

```c
int fibseq(int n)  
int fibpar(int n)
```

We require the user to provide an asymptotic cost function:

```c
int fibcost(int n) {
    return 1.618 ** n;
}
```

We use runtime profiling to deduce the associated constant factor

Benefits:
→ complexity annotations are hardware independent
→ runtime profiling does not impose a per-algorithm tuning phase

Limitations:
→ cost functions must be cheap to evaluate
→ average complexity needs to match worst-case complexity
source code:

```c
int fib(int n) {
    costs { return 1.618 ** n; }
    if (n < 2) return 1;
    spawn int a = fib(n-1);
    int b = fib(n-2);
    sync;
    return a+b;
}
```

compiled into:

- `int fibseq(int n)`
- `int fibpar(int n)`
- `int fibcost(int n)`

(translation implemented for the ML front-end, not yet for the C front-end)
Convergence of the constant

- Pessimistic start
- Exponential decrease
- Fast convergence
- Continue measuring

value of the constant (in microsecond/unit), on a log scale

timeline of the program execution (seconds)
Accuracy of the predictions

(measured on the cilksort benchmark)
Theory, generalized model

– let $\phi$ be the cost of making a time prediction and a time measure
– let $\mu$ be the maximal error factor for predictions
– let $\gamma$ the max ratio between two time predictions ($\gamma=2$ for most programs)

\[
T_P \leq \left(1 + \frac{\mu(\tau + \gamma\phi)}{\kappa}\right) \cdot \frac{T_1}{P} + (\kappa \mu + \phi + 1) \cdot T_\infty
\]

Example:

$\tau = 100$ ns  \quad $\mu = 2$  \quad $T_1 = 10^9 \cdot 10$ns

$\phi = 200$ ns  \quad $\gamma = 2$  \quad $T_\infty = 30$

$\kappa = 100,000$ ns ($= 0.1$ms)  \quad $P = 30$

just a few percent  \quad relatively small

1%  \quad 2% of the first term
Benchmarks

**Benchmarks:** quickhull, quicksort, barnes-hut, dense matrix multiply, sparse matrix multiply, KMP string search, Bellman-Ford algorithm, ...

**Examples of complexity functions:**

```
return 1.618 ** n
return n * log n
return n ** 3
return high - low
return prefixsum[high] - prefixsum[low]
```

**Results:**

→ appropriate cutoff values are selected
→ the overheads do not exceed a few percents
Speedup curve: fib of 45

→ selected cutoff = 20

good speedup, though not all processors busy at the end

insignificant task creation overheads
Speedup curve: cilksort on $10^8$ integers

→ selected cutoff $\approx 13,000$ items

AMD, NUMA, 8 nodes with 6 cores each, 2Ghz

INTEL, UMA, 4 nodes with 8 cores each, 2Ghz
Speedup curve: KMP on $10^9$ chars

AMD, NUMA, 8 nodes with 6 cores each, 2Ghz

→ speedups achieved without tuning phase nor hardcoding of the cutoff
Conclusion

1) asymptotic complexity annotations + runtime profiling
   → enable execution time predictions

2) sequentializing all tasks that are predicted to be small
   → ensure that task creation overheads are well amortized

Good granularity control with little effort!

Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages
Umut A. Acar, Arthur Charguéraud and Mike Rainey